

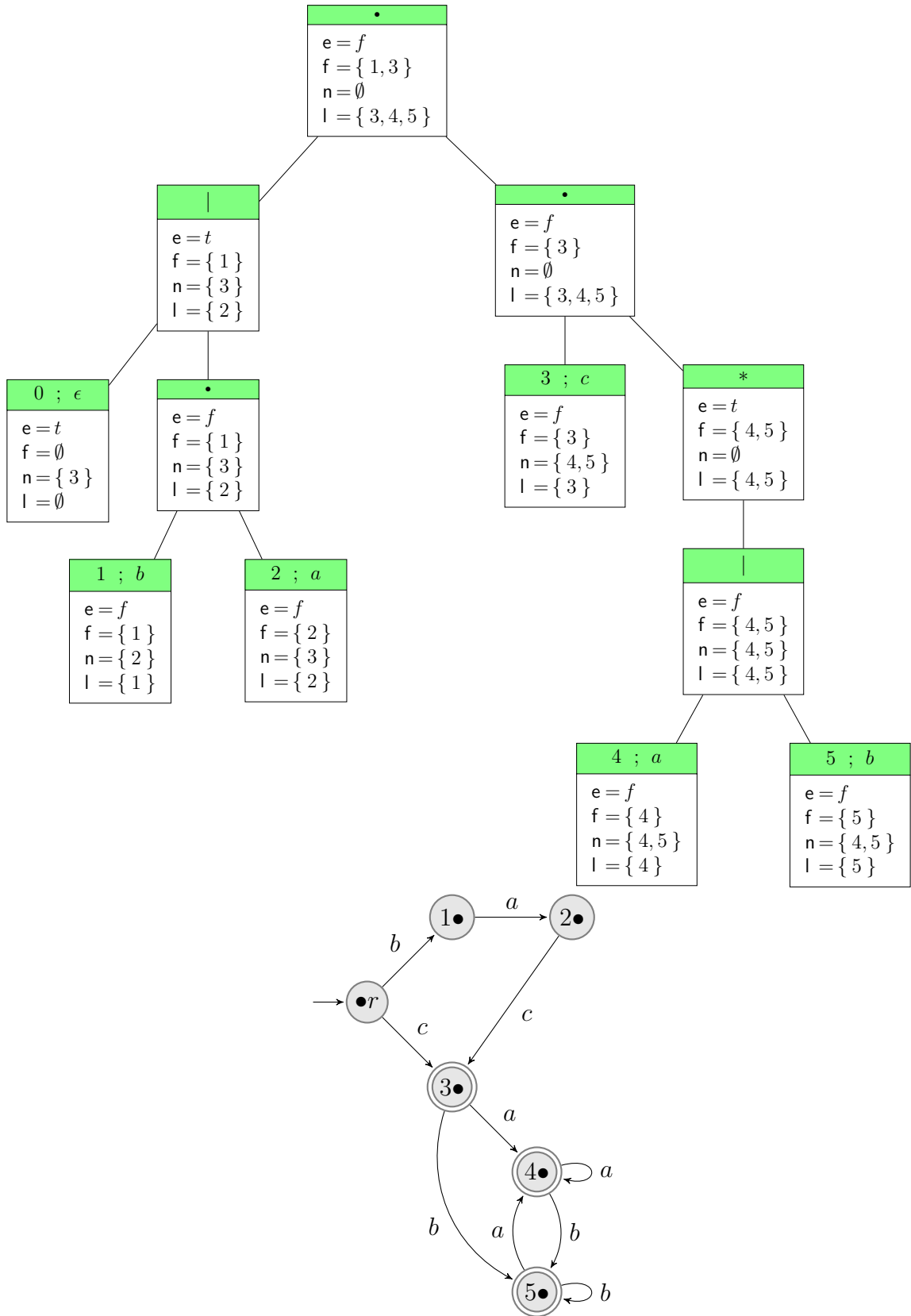
## Exercise Sheet 2

### Assignment 2.1 Berry-Sethi Algorithm (Sophisticated Approach)

Use the Berry-Sethi Algorithm and transform the expression  $r = (\epsilon|ba)(c(a|b)^*)$  as follows:

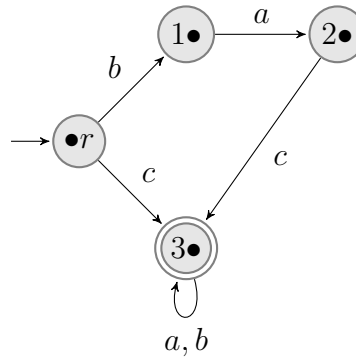
1. Draw the regular expression as a tree.
2. Compute the **empty** attribute.
3. Compute the **first** attribute.
4. Compute the **next** attribute.
5. Compute the **last** attribute.
6. Construct and draw the automaton.
7. Is the resulting automaton deterministic or not?
8. Is the resulting automaton minimal or not?

### Suggested Solution 2.1



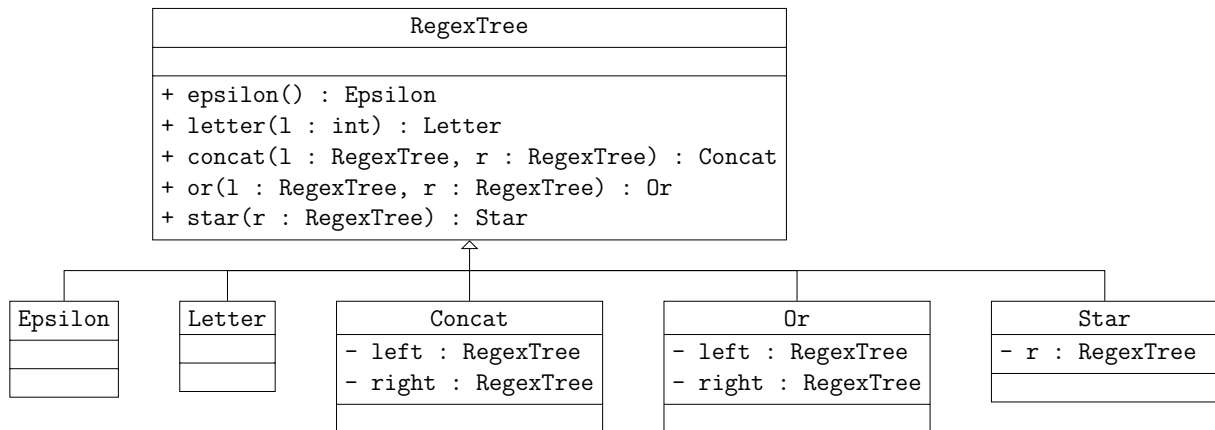
The automaton is deterministic.

The automaton is not minimal. The states  $3 \bullet$ ,  $4 \bullet$ , and  $5 \bullet$  can be merged into one state:



## Assignment 2.2 Berry-Sethi is alive!

Get your hands dirty! Basically your task is to implement the sophisticated Berry-Sethi approach in Java. In order to do so we have to represent a regular expression as a tree. You might want to start with the following class layout:



We represent elements from our alphabet  $\Sigma$  by objects of the class **Letter** and assume that our alphabet  $\Sigma$  are all unicode characters.

So far, we cannot properly parse (we will change this in the following weeks). Therefore, as an input we do not parse a regular expression given as a string and construct our tree from that, instead, we construct the tree directly via helper methods defined in the class **RegexTree**. For example, the following call

```
concat(star(or(letter('a'), letter('b'))),
      concat(letter('a'),
            or(letter('a'), letter('b'))))
```

returns a corresponding tree of the regular expression  $(a|b)^*(a(a|b))$ .

Extend the classes to suit your needs. Compute the attributes **empty**, **first**, **next**, and **last**. Hint: You can traverse the tree by directly implementing corresponding methods in the tree-classes, or, you can implement the visitor-pattern. The latter approach is typically the better way to do the job (in a few weeks you will learn the visitor-pattern in class anyway).

Now we want to visualize the non-deterministic finite automata. Make use of the Graphviz DOT language in order to visualize the automata. A corresponding NFA from the example regular expression from above is given in the DOT language as follows:

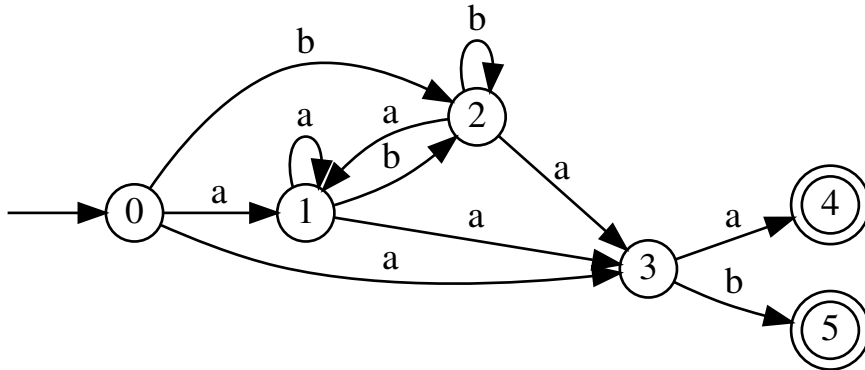
```
digraph nfa {
  rankdir=LR;
```

```

size="8,5"
node [shape=none,width=0,height=0,margin=0]; start [label=""];
node [shape=doublecircle];
4;5;
node [shape=circle];
0 -> 1 [label="a"];
0 -> 3 [label="a"];
0 -> 2 [label="b"];
1 -> 1 [label="a"];
1 -> 3 [label="a"];
1 -> 2 [label="b"];
3 -> 4 [label="a"];
3 -> 5 [label="b"];
2 -> 1 [label="a"];
2 -> 3 [label="a"];
2 -> 2 [label="b"];
start -> 0;
}

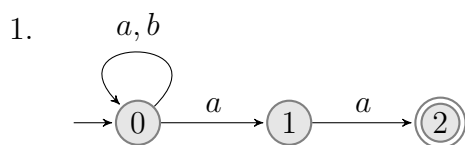
```

Assuming that the content from above is saved in a file named `nfa.gv`, you can then layout the automata via the command `dot -Tpdf nfa.gv -O` resulting in a PDF file `nfa.gv.pdf`:

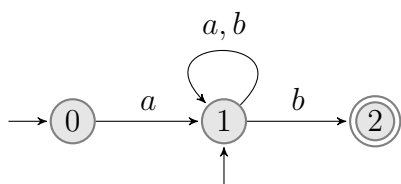


### Assignment 2.3 Powerset Construction

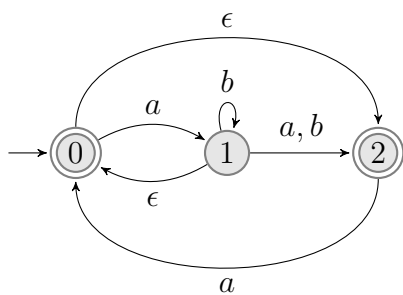
Transform the following NFAs to DFAs using the powerset construction. If necessary add a  $\emptyset$ -state.



2.

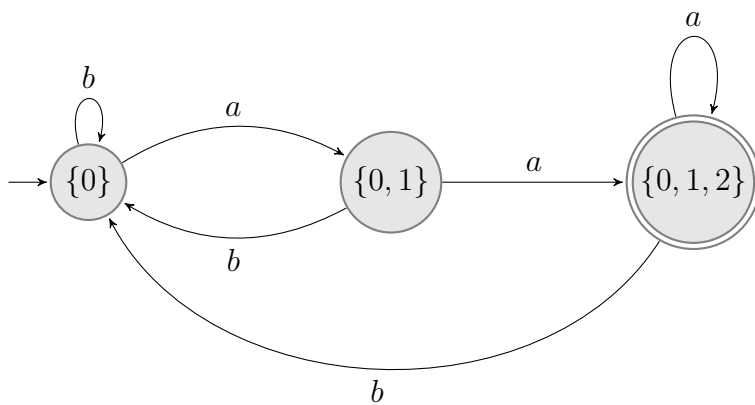


3.

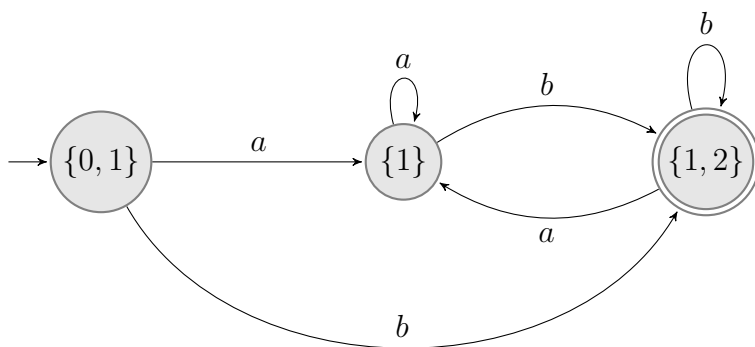


### Suggested Solution 2.3

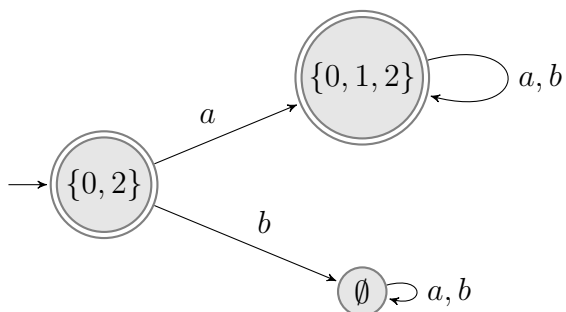
1.



2.

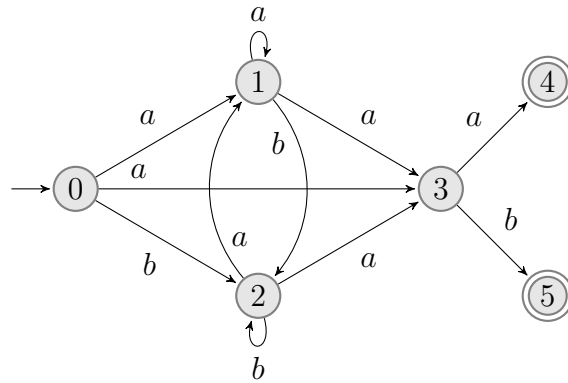


3.



### Assignment 2.4 Partial Powerset Construction

Consider the following NFA from the lecture:



Construct only the part of the corresponding DFA that is needed for the input  $baaa$ .

### Suggested Solution 2.4

